

Classical programming languages cannot model essential elements of complex systems such as true random number generation. This paper develops a formal programming language called the lambda-q calculus that addresses the fundamental properties of complex systems. This formal language allows the expression of quantumized algorithms, which are extensions of randomized algorithms in that probabilities can be negative, and events can cancel out. An illustration of the power of quantumized algorithms is the ability to efficiently solve the satisfiability problem, something that many believe is beyond the capability of classical computers. This paper proves that the lambda-q calculus is not only capable of solving satisfiability but can also simulate such complex systems as quantum computers. Since satisfiability is believed to be beyond the capabilities of quantum computers, the lambda-q calculus may be strictly stronger.

## 1. Introduction

The purpose of this paper is to introduce a formalism for expressing models of complex systems. The end result is that modelling any complex system such as human society, evolution, or particle interactions, may be reduced to a programming problem.

In addition to the modelling functionalities it provides, a programmable complex system also allows us to see, in its specification, what the distilled and essential elements of a complex system are. In particular, as we will see, interactions like those in a cellular automaton need not be explicit in the formalism, as they may be simulated.

Classical programming languages are not strong enough to model complex systems. They do not allow for randomized events and are completely predictable and deterministic, features rarely found in complex systems. Some problems that may be quickly solved on quantum computers, which is a complex system, have no known quick solutions on classical computers or with classical programming languages.

In this paper we extend the  $\lambda$ -calculus, the logical foundation of classical programming languages. The first extension, the  $\lambda^p$ -calculus, is a new calculus introduced here for expressing randomized functions. Randomized functions, instead of having a unique output for each input, return a distribution of results from which we sample once. The  $\lambda^p$ -calculus then provides a formal method for computing distributions. More useful, however, would be the ability to compute conditional distributions. The second extension, the  $\lambda^q$ -calculus, is a new calculus introduced here for expressing quantumized functions. Quantumized functions also return a distribution of results, called a *superposition*, from which we sample once, but  $\lambda^q$ -terms have signs, and identical terms with opposite signs are removed before sampling from the result. Quantumized functions can then compute conditional distributions. The effect is that of applying some filter to a superposition to adjust each of the probabilities according to its fitness. One example is the quick

solution of satisfiability: by merely filtering out the logical mappings of variables that do not satisfy the given formula, we are left only with satisfying mappings, if any. The  $\lambda^q$ -calculus is the most general of the three calculi.

One of the results of this paper is that the  $\lambda^q$ -calculus is at least as powerful as quantum computers. Although much research has been done on the hardware of quantum computation (c.f. [5], [6], [10]), none has focused on formalizing the software. Quantum Turing machines [5] have been introduced but there has been no quantum analogue to Church's  $\lambda$ -calculus. The  $\lambda$ -calculus has served as the basis for many programming languages since it was introduced by Alonzo Church [4] in 1936. It and other classical calculi make the implicit assumption that a term may be innocuously observed at any point. Such an assumption is hard to separate from a system of rewriting rules because to rewrite a term, you must have read it. One of the goals of these calculi is to make observation explicit.

The  $\lambda^p$ - and the  $\lambda^q$ -calculi allow the expression of algorithms that exist and operate in the Heisenberg world of *potentia* [7] but whose results are observed. To this end, collections (distributions and superpositions) should be thought of with the following intuition. A collection is a bunch of terms that co-exist in the same place but are not aware of each other. Thus, a collection of three terms takes up no more space than a collection of two terms. A physical analogy is the ability of a particle to be in a superposition of states. When the collection is observed, at most one term in each collection will be the result of the observation. The key point is that in neither calculus can one write a term that can determine if it is part of a collection, how big the collection is, or even if its argument is part of a collection. Despite this inability, the  $\lambda^q$ -calculus is powerful enough to efficiently solve problems such as satisfiability that are typically believed to be beyond the scope of classical computers.

## 2. The Lambda Calculus

This section is a review of the  $\lambda$ -calculus and a reference for later calculi. For more details see e.g. [1].

The  $\lambda$ -calculus is a calculus of functions. Any computable single-argument function can be expressed in the  $\lambda$ -calculus. Any computable multiple-argument function can be expressed in terms of computable single-argument functions. The  $\lambda$ -calculus is useful for encoding functions of arbitrary arity that return at most one output for each input. In particular, the  $\lambda$ -calculus can be used to express any (computable) *algorithm*. The definition of algorithm is usually taken to be Turing-computable.

### 2.1. Syntax

The following grammar specifies the syntax of the  $\lambda$ -calculus.

$x$	$\in$ Variable	Variables
$M$	$\in$ LambdaTerm	Terms
$w$	$\in$ Wff	Well-formed formulas
$M$	$::=$ $x$	variable
	$ $ $M_1 M_2$	application
	$ $ $\lambda x.M$	abstraction
$w$	$::=$ $M_1 = M_2$	well-formed formula

(2.1)

To be strict, the subscripts above should be removed (e.g., the rule for well-formed formulas should read  $w ::= M = M$ ) because  $M_1$  and  $M_2$  are not defined. However, we will maintain this incorrect notation to emphasize that the terms need not be identical.

With this abuse of notation, we can easily read the preceding definition as: a  $\lambda$ -term is a variable, or an application of two terms, or the abstraction of a term by a variable. A well-formed formula of the  $\lambda$ -calculus is a  $\lambda$ -term followed by the equality sign followed by a second  $\lambda$ -term.

We also adopt some syntactic conventions. Most importantly, parentheses group subexpressions. Application is taken to be left associative so that the term  $MNP$  is correctly parenthesized as  $(MN)P$  and not as  $M(NP)$ . The scope of an abstraction extends as far to the right as possible, for example up to a closing parenthesis, so that the term  $\lambda x.xx$  is correctly parenthesized as  $(\lambda x.xx)$  and not as  $(\lambda x.x)x$ .

### 2.2. Substitution

We will want to substitute arbitrary  $\lambda$ -terms for variables. We define the substitution operator, notated  $M[N/x]$  and read “ $M$  with all free occurrences of  $x$  replaced by  $N$ .” The definition of the free and bound variables of a term are standard. The set of free variables of a term  $M$  is written  $FV(M)$ . There are six rules of substitution, which we write

for reference.

1.  $x[N/x] \equiv N$
2.  $y[N/x] \equiv y$  for variables  $y \not\equiv x$
3.  $(PQ)[N/x] \equiv (P[N/x])(Q[N/x])$
4.  $(\lambda x.P)[N/x] \equiv \lambda x.P$
5.  $(\lambda y.P)[N/x] \equiv \lambda y.(P[N/x])$  if  $y \not\equiv x$  and  $y \notin FV(N)$
6.  $(\lambda y.P)[N/x] \equiv \lambda z.(P[z/y][N/x])$   
 $y \not\equiv x,$   
if  $y \in FV(N)$ , and  
 $z \notin FV(P) \cup FV(N)$

(2.2)

This definition will be extended in both subsequent calculi.

### 2.3. Reduction

The concept of *reduction* seeks to formalize rewriting rules. Given a relation  $R$  between terms, we may define the one-step reduction relation, notated  $\rightarrow_R$ , that is the contextual closure of  $R$ . We may also define the reflexive, transitive closure of the one-step reduction relation, which we call  $R$ -reduction and notate  $\rightarrow_R^*$ , and the symmetric closure of  $R$ -reduction, called  $R$ -interconvertibility and notated  $=_R$ .

The essential notion of reduction for the  $\lambda$ -calculus is called  $\beta$ -reduction. It is based on the  $\beta$ -relation, which is the formalization of function invocation.

$$\beta \triangleq \left\{ \begin{array}{l} ((\lambda x.M) N, M[N/x]) \\ \text{s.t. } M, N \in \text{LambdaTerm}, x \in \text{Variable} \end{array} \right\} \quad (2.3)$$

There is also the  $\alpha$ -relation that holds of terms that are identical up to a consistent renaming of variables.

$$\alpha \triangleq \left\{ \begin{array}{l} (\lambda x.M, \lambda y.M[y/x]) \\ \text{s.t. } M \in \text{LambdaTerm}, y \notin FV(M) \end{array} \right\} \quad (2.4)$$

We will use this only sparingly.

### 2.4. Evaluation Semantics

By imposing an evaluation order on the reduction system, we are providing meaning to the  $\lambda$ -terms. The evaluation order of a reduction system is sometimes called an operational semantics or an evaluation semantics for the calculus. The evaluation relation is typically denoted  $\rightsquigarrow$ .

We use call-by-value evaluation semantics. A *value* is the result produced by the evaluation semantics. Call-by-value semantics means that the body of an abstraction is not reduced but arguments are evaluated before being passed into abstractions.

There are two rules for the call-by-value evaluation semantics of the  $\lambda$ -calculus.

$$\frac{}{v \rightsquigarrow v} (\text{Refl}) \quad (\text{for } v \text{ a value})$$

$$\frac{M \rightsquigarrow \lambda x.P \quad N \rightsquigarrow N' \quad P[N'/x] \rightsquigarrow v}{MN \rightsquigarrow v} (\text{Eval})$$

## 2.5. Reference Terms

The following  $\lambda$ -terms are standard and are provided as reference for later examples.

Numbers are represented as Church numerals.

$$\underline{0} \equiv \lambda x. \lambda y. y \quad (2.5)$$

$$\underline{n} \equiv \lambda x. \lambda y. x^n y \quad (2.6)$$

where the notation  $x^n y$  means  $n$  right-associative applications of  $x$  onto  $y$ . It is abbreviatory for the term  $\underbrace{x(x \cdots (x y))}_{n \text{ times}}$ . When necessary, we can extend Church

numerals to represent both positive and negative numbers. For the remainder of the terms, we will not provide definitions. The predecessor of Church numerals is written  $\underline{P}$ . The successor is written  $\underline{S}$ .

The conditional is written  $\underline{IF}$ . If its first argument is truth, written  $\underline{T}$ , then it returns its second argument. If its first argument is falsity, written  $\underline{F}$ , then it returns its third argument. A typical predicate is  $\underline{0?}$  which returns  $\underline{T}$  if its argument is the Church numeral  $\underline{0}$  and  $\underline{F}$  if it is some other Church numeral.

The fixed-point combinator is written  $\underline{Y}$ . The primitive recursive function-building term is written  $\underline{\text{PRIM-REC}}$  and it works as follows. If the value of a function  $f$  at input  $n$  can be expressed in terms of  $n - 1$  and  $f(n - 1)$ , then that function  $f$  is primitive recursive, and it can be generated by providing  $\underline{\text{PRIM-REC}}$  with the function that takes the inputs  $n - 1$  and  $f(n - 1)$  to produce  $f(n)$  and with the value of  $f$  at input 0. For example, the predecessor function for Church numerals can be represented as  $\underline{P} \equiv \underline{\text{PRIM-REC}} (\lambda x. \lambda y. x) \underline{0}$ .

## 3. The Lambda-P Calculus

The  $\lambda^p$ -calculus is an extension of the  $\lambda$ -calculus that permits the expression of *randomized* algorithms. In contrast with a computable algorithm which returns at most one output for each input, a randomized algorithm returns a *distribution* of answers from which we sample. There are several advantages to randomized algorithms.

1. Randomized algorithms can provide truly random number generators instead of relying on pseudo-random number generators that work only because the underlying pattern is difficult to determine.
2. Because they can appear to generate random numbers arbitrarily, randomized algorithms can model random processes.
3. Given a problem of finding a suitable solution from a set of possibilities, a randomized algorithm can exhibit the effect of choosing random elements and testing them. Such algorithms can sometimes have an *expected* running time which is considerably shorter than the running time of the computable algorithm that tries every possibility until it finds a solution.

## 3.1. Syntax

The following grammar describes the  $\lambda^p$ -calculus.

$x$	$\in$ Variable	Variables
$M$	$\in$ LambdaPTerm	Terms
$w$	$\in$ WffP	Well-formed formulas
<hr/>		
$M$	$::=$ $x$	variable
	$ $ $M_1 M_2$	application
	$ $ $\lambda x. M$	abstraction
	$ $ $M_1, M_2$	collection
<hr/>		
$w$	$::=$ $M_1 = M_2$	well-formed formula

(3.1)

Since this grammar differs from the  $\lambda$ -calculus only in the addition of the fourth rule for terms, all  $\lambda$ -terms can be viewed as  $\lambda^p$ -terms. A  $\lambda^p$ -term may be a collection of a term and another collection, so that a  $\lambda^p$ -term may actually have many nested collections.

We adhere to the same parenthesization and precedence rules as the  $\lambda$ -calculus with the following addition: collection is of lowest precedence and the comma is right associative. This means that the expression  $\lambda x. x, z, y$  is correctly parenthesized as  $(\lambda x. x), (z, y)$ .

We introduce abbreviatory notation for collections. Let us write  $[M_i^{i \in S}]$  for the collection of terms  $M_i$  for all  $i$  in the finite, ordered set  $S$  of natural numbers. We will write  $a..b$  for the ordered set  $(a, a + 1, \dots, b)$ . In particular,  $[M_i^{i \in 1..n}]$  represents  $M_1, M_2, \dots, M_n$  and  $[M_i^{i \in n..1}]$  represents  $M_n, M_{n-1}, \dots, M_1$ . More generally, let us allow multiple iterators in arbitrary contexts. Then, for instance,

$$[\lambda x. M_i^{i \in 1..n}] \equiv \lambda x. M_1, \lambda x. M_2, \dots, \lambda x. M_n$$

and

$$[M_i^{i \in 1..m} N_j^{j \in 1..n}] \equiv \begin{array}{c} M_1 N_1, M_1 N_2, \dots, M_1 N_n, \\ M_2 N_1, M_2 N_2, \dots, M_2 N_n, \\ \vdots \\ M_m N_1, M_m N_2, \dots, M_m N_n \end{array}.$$

Note that  $[\lambda x. M_i^{i \in 1..n}]$  and  $\lambda x. [M_i^{i \in 1..n}]$  are not the same term. The former is a collection of abstractions while the latter is an abstraction with a collection in its body. Finally, we allow this notation to hold of non-collection terms as well by identifying  $[M_i^{i \in 1..1}]$  with  $M_1$  even if  $M_1$  is not a collection. To avoid confusion, it is important to understand that although this “collection” notation can be used for non-collections, we do not extend the definition of the word *collection*. A *collection* is still the syntactic structure defined in grammar (3.1).

With these additions, every term can be written in this bracket form. In particular, we can write a collection as  $[M_i^{i \in S_i}]_j^{j \in S}$ , or a collection of collections. Unfortunately, collections can be written in a variety of ways with this notation. The term  $M, N, P$  can be written as  $[M_i^{i \in 1..3}]$  if  $M_1 \equiv M$  and  $M_2 \equiv N$  and  $M_3 \equiv P$ ; as  $[M_i^{i \in 1..2}]$  if  $M_1 \equiv M$

and  $M_2 \equiv N, P$ ; or as  $[M_i^{i \in 1..1}]$  if  $M_1 \equiv M, N, P$ . However, it cannot be written as  $[M_i^{i \in 1..4}]$  for any identification of the  $M_i$ . This observation inspires the following definition.

**Definition 1** *The cardinality of a term  $M$ , notated  $|M|$ , is that number  $k$  for which  $[M_i^{i \in 1..k}] \equiv M$  for some identification of the  $M_i$  but  $[M_i^{i \in 1..(k+1)}] \not\equiv M$  for any identification of the  $M_i$ .*

Note that the cardinality of a term is always strictly positive.

### 3.2. Syntactic Identities

We define substitution of terms in the  $\lambda^p$ -calculus as an extension of substitution of terms in the  $\lambda$ -calculus. In addition to the substitution rules of the  $\lambda$ -calculus, we introduce one for collections.

$$(P, Q) [N/x] \equiv (P [N/x], Q [N/x]) \quad (3.2)$$

We identify terms that are collections but with a possibly different ordering. We also identify nested collections with the top-level collection. The motivation for this is the conception that a collection is an unordered set of terms. Therefore we will not draw a distinction between a set of terms and a set of a set of terms.

We adopt the following axiomatic judgement rules.

$$\frac{}{M, N \equiv N, M} (\text{ClnOrd})$$

$$\frac{}{(M, N), P \equiv M, (N, P)} (\text{ClnNest})$$

With these axioms, ordering and nesting become innocuous. As an example here is the proof that  $A, (B, C), D \equiv A, C, B, D$ . For clarity, we parenthesize fully and underline the affected term in each step.

$$\begin{aligned} \underline{A, ((B, C), D)} &\equiv ((\underline{B, C}), D), A & (\text{ClnOrd}) \\ &\equiv ((\underline{C, B}), D), A & (\text{ClnOrd}) \\ &\equiv (\underline{C, (B, D)}), A & (\text{ClnNest}) \\ &\equiv A, (C, (B, D)) & (\text{ClnOrd}) \end{aligned}$$

It can be shown that ordering and parenthesization are irrelevant in general. Aside, it no longer matters that we took the comma to be right associative since any arbitrary parenthesization of a collection does not change its syntactic structure.

Because of this theorem, we can alter the abbreviatory notation and allow arbitrary unordered sets in the exponent. This allows us to write, for instance,  $[M_i^{i \in 1..n - \{j\}}] \equiv M_1, M_2, \dots, M_{j-1}, M_{j+1}, \dots, M_n$  where  $a..b$  is henceforth taken to be the unordered set  $\{a, a+1, \dots, b\}$  and the subtraction in the exponent represents set difference.

This also subtly alters the definition of *cardinality* (1). Whereas before the cardinality of a term like  $(x, y), z$  was 2, because of this theorem, it is now 3.

We may now also introduce a further abbreviation. We let  $[(M_i : n_i)]$  be a rewriting of the term  $[N_i^{i \in I}]$  such each of the  $M_i$  are distinct and the integer  $n_i$  represents the count of each  $M_i$  in  $[N_i^{i \in I}]$ .

### 3.3. Reductions

The relation of collection application is called the  $\gamma$ -relation. It holds of a term that is an application at least one of whose operator or operand is a collection, and the term that is the collection of all possible pairs of applications.

$$\gamma^p \triangleq \left\{ \begin{array}{l} ([M_i^{i \in 1..m}] [N_j^{j \in 1..n}], [M_i^{i \in 1..m} N_j^{j \in 1..n}]) \\ \text{s.t. } M_i, N_j \in \text{LambdaPTerm}, m > 1 \text{ or } n > 1 \end{array} \right\} \quad (3.3)$$

We will omit the superscript except to disambiguate from the  $\gamma$ -relation of the  $\lambda^q$ -calculus.

It can be shown that the  $\gamma$ -relation is Church-Rosser and that all terms have  $\gamma$ -normal forms. Therefore, we may write  $\gamma(M)$  for the  $\gamma$ -normal form of  $M$ .

We extend the  $\beta$ -relation to apply to collections.

$$\beta^p \triangleq \left\{ \begin{array}{l} ((\lambda x. M) [N_i^{i \in S}], [M [N_i^{i \in S}/x]]) \\ \text{s.t. } M, [N_i^{i \in S}] \in \text{LambdaPTerm}, x \in \text{Variable} \end{array} \right\} \quad (3.4)$$

where  $[M [N_i^{i \in S}/x]]$  is the collection of terms  $M$  with  $N_i$  substituted for free occurrences of  $x$  in  $M$ , for  $i \in S$ .

### 3.4. Evaluation Semantics

We extend the call-by-value evaluation semantics of the  $\lambda$ -calculus. We modify the definition of a value  $v$  to enforce that  $v$  has no  $\gamma$ -redexes.

$$\begin{aligned} &\frac{}{v \rightsquigarrow v} (\text{Refl}) & (\text{for } v \text{ a value}) \\ &\frac{\gamma(M) \rightsquigarrow \lambda x. P \quad \gamma(N) \rightsquigarrow N' \quad \gamma(P [N'/x]) \rightsquigarrow v}{MN \rightsquigarrow v} (\text{Eval}) \\ &\frac{\gamma(M) \rightsquigarrow v_1 \quad \gamma(N) \rightsquigarrow v_2}{(M, N) \rightsquigarrow (v_1, v_2)} (\text{Coll}) \end{aligned}$$

### 3.5. Observation

We define an observation function  $\Theta$  from  $\lambda^p$ -terms to  $\lambda$ -terms. We employ the random number generator  $RAND$ , which samples one number from a given set of numbers.

$$\Theta(x) = x \quad (3.5)$$

$$\Theta(\lambda x. M) = \lambda x. \Theta(M) \quad (3.6)$$

$$\Theta(M_1 M_2) = \Theta(M_1) \Theta(M_2) \quad (3.7)$$

$$\Theta(M \equiv [M_i^{i \in 1..|M|}]) = M_{RAND(1..|M|)} \quad (3.8)$$

The function  $\Theta$  is total because every  $\lambda^p$ -term is mapped to a  $\lambda$ -term. Note that for an arbitrary term  $T$  we may write  $\Theta(T) = T_{RAND(S)}$  for some possibly singleton set of natural numbers  $S$  and some collection of terms  $[T_i^{i \in S}]$ .

We can show that observing a  $\lambda^p$ -term is statistically indistinguishable from observing its  $\gamma$ -normal form.

### 3.6. Observational Semantics

We provide another type of semantics for the  $\lambda^p$ -calculus called its *observational semantics*. A formalism's observational semantics expresses the computation as a whole: preparing the input, waiting for the evaluation, and observing the result. The observational semantics relation between  $\lambda^p$ -terms and  $\lambda$ -terms is denoted  $\multimap$ . It is given by a single rule for the  $\lambda^p$ -calculus.

$$\frac{M \rightsquigarrow v \quad \Theta(v) = N}{M \multimap N} (\text{ObsP}) \quad (3.9)$$

### 3.7. Examples

A useful term of the  $\lambda^p$ -calculus is a random number generator. We would like to define a term that takes as input a numeral  $\underline{n}$  and computes a collection of numerals from  $\underline{0}$  to  $\underline{n}$ . This can be represented by the following primitive recursive  $\lambda^p$ -term.

$$\underline{R} \equiv \underline{\text{PRIM-REC}} (\lambda k. \lambda p. (k, p)) \underline{0} \quad (3.10)$$

Then for instance  $\underline{R} \underline{3} = (\underline{3}, \underline{2}, \underline{1}, \underline{0})$ .

The following term represents a random walk. Imagine a man that at each moment can either walk forward one step or backwards one step. If he starts at the point 0, after  $n$  steps, what is the distribution of his position?

$$\underline{W} \equiv \underline{\text{PRIM-REC}} (\lambda k. \lambda p. (\underline{P}p, \underline{S}p)) \underline{0} \quad (3.11)$$

We assume we have extended Church numerals to negative numbers as well. This can be easily done by encoding it is a pair. We will show some of the highlights of the evaluation of  $\underline{W} \underline{3}$ . Note that  $\underline{W} \underline{1} = (\underline{-1}, \underline{1})$ .

$$\begin{aligned} \underline{W} \underline{3} &= \underline{P}(\underline{W} \underline{2}), \underline{S}(\underline{W} \underline{2}) \\ &= \underline{P}(\underline{P}(\underline{W} \underline{1}), \underline{S}(\underline{W} \underline{1})), \underline{S}(\underline{P}(\underline{W} \underline{1}), \underline{S}(\underline{W} \underline{1})) \\ &= \underline{P}(\underline{P}(\underline{-1}, \underline{1}), \underline{S}(\underline{-1}, \underline{1})), \underline{S}(\underline{P}(\underline{-1}, \underline{1}), \underline{S}(\underline{-1}, \underline{1})) \\ &= \underline{P}((\underline{-2}, \underline{0}), (\underline{0}, \underline{2})), \underline{S}((\underline{-2}, \underline{0}), (\underline{0}, \underline{2})) \\ &= ((\underline{-3}, \underline{-1}), (\underline{-1}, \underline{1})), ((\underline{-1}, \underline{1}), (\underline{1}, \underline{3})) \\ &\equiv (\underline{-3}, \underline{-1}, \underline{-1}, \underline{1}, \underline{-1}, \underline{1}, \underline{1}, \underline{3}) \end{aligned} \quad (3.12)$$

Observing  $\underline{W} \underline{3}$  yields  $\underline{-1}$  with probability  $\frac{3}{8}$ ,  $\underline{1}$  with probability  $\frac{3}{8}$ ,  $\underline{-3}$  with probability  $\frac{1}{8}$ , and  $\underline{3}$  with probability  $\frac{1}{8}$ .

## 4. The Lambda-Q Calculus

The  $\lambda^q$ -calculus is an extension of the  $\lambda^p$ -calculus that allows easy expression of *quantumized* algorithms. A quantumized algorithm differs from a randomized algorithm in allowing negative probabilities and in the way we sample from the resulting distribution.

Variables and abstractions in the  $\lambda^q$ -calculus have *phase*. The phase is nothing more than a plus or minus sign, but since the result of a quantumized algorithm is a distribution of terms with phase, we call such a distribution by the special name *superposition*. The major difference between

a superposition and a distribution is the observation procedure. Before randomly picking an element, a superposition is transformed into a distribution by the following two-step process. First, all terms in the superposition that are identical except with opposite phase are cancelled. They are both simply removed from the superposition. Second, the phases are stripped to produce a distribution. Then, an element is chosen from the distribution randomly, as in the  $\lambda^p$ -calculus.

The words *phase* and *superposition* come from quantum physics. An electron is in a superposition if it can be in multiple possible states. Although the phases of the quantum states may be any angle from  $0^\circ$  to  $360^\circ$ , we only consider binary phases. Because we use solely binary phases, we will use the words *sign* and *phase* interchangeably in the sequel.

A major disadvantage of the  $\lambda^p$ -calculus is that it is impossible to compress a collection. Every reduction step at best keeps the collection the same size. Quantumized algorithms expressed in the  $\lambda^q$ -calculus, on the other hand, can do this as easily as randomized algorithms can generate random numbers. That is,  $\lambda^q$ -terms can contain subterms with opposite signs which will be removed during the observation process.

### 4.1. Syntax

The following grammar describes the  $\lambda^q$ -calculus.

$S$	$\in \text{Sign}$	Sign, or phase
$x$	$\in \text{Variable}$	Variables
$M$	$\in \text{LambdaQTerm}$	Terms
$w$	$\in \text{WffQ}$	Well-formed formulas
$S$	$::= +$ $\quad   -$	positive negative
$M$	$::= Sx$ $\quad   M_1 M_2$ $\quad   S\lambda x. M$ $\quad   M_1, M_2$	signed variable application signed abstraction collection
$w$	$::= M_1 = M_2$	well-formed formula

(4.1)

Terms of the  $\lambda^q$ -calculus differ from terms of the  $\lambda^p$ -calculus only in that variables and abstractions are *signed*, that is, they are preceded by either a plus (+) or a minus (-) sign. Just as  $\lambda$ -terms could be read as  $\lambda^p$ -terms, we would like  $\lambda^p$ -terms to be readable as  $\lambda^q$ -terms. However,  $\lambda^p$ -terms are unsigned and cannot be recognized by this grammar.

Therefore, as is traditionally done with integers, we will omit the positive sign. An unsigned term in the  $\lambda^q$ -calculus is abbreviatory for the same term with a positive sign. With this convention,  $\lambda^p$ -terms can be seen as  $\lambda^q$ -terms all of whose signs are positive. Also, so as not to confuse a negative sign with subtraction, we will write it with a logical negation sign ( $\neg$ ). With these two conventions, the  $\lambda^q$ -term  $+\lambda x. +x - x$  is written simply  $\lambda x. x \neg x$ .

Finally, we adhere to the same parenthesization and precedence rules as the  $\lambda^p$ -calculus. In particular, we continue the use of the abbreviatory notations  $[M_i^{i \in S}]$  and  $[(M_i : n_i)]$  for collections of terms. In addition, we can also  $[(M_i : n_i)]$  as  $[(M_i : a_i, b_i, n_i)]$  such that  $M_i \not\equiv M_j$  and  $M_i \equiv \overline{M_j}$  for  $i \neq j$ , all of the  $M_i$  are of positive sign, the integer  $a_i$  denotes the count of  $M_i$ , the integer  $b_i$  denotes the count of  $\overline{M_i}$ , and  $n_i = a_i - b_i$ .

#### 4.2. Syntactic Identities

We will call two terms *opposites* if they differ only in sign.

We define substitution of terms in the  $\lambda^q$ -calculus as a modification of substitution of terms in the  $\lambda^p$ -calculus. We rewrite the seven rules of the  $\lambda^p$ -calculus to take account of the signs of the terms. First, we introduce the function notated by sign concatenation, defined by the following rule in our abbreviatory conventions.

$$\neg\neg \mapsto \epsilon \quad (4.2)$$

We also note that the concatenation of a sign  $S$  with  $\epsilon$  is just  $S$  again. Now we can use this function in the following substitution rules.

1.  $(Sx)[N/x] \equiv Sx$
  2.  $(Sy)[N/x] \equiv Sy$  for variables  $y \not\equiv x$
  3.  $(PQ)[N/x] \equiv (P[N/x])(Q[N/x])$
  4.  $(S\lambda x.P)[N/x] \equiv S\lambda x.P$
  5.  $(S\lambda y.P)[N/x] \equiv S\lambda y.(P[N/x])$  if  $y \not\equiv x$ , and  $y \notin FV(N)$
  6.  $(S\lambda y.P)[N/x] \equiv S\lambda z.(P[z/y][N/x])$   
 $y \not\equiv x$ ,  
if  $y \in FV(N)$ , and  
 $z \notin FV(P) \cup FV(N)$
  7.  $(P, Q)[N/x] \equiv (P[N/x], Q[N/x])$
- (4.3)

The use of the sign concatenation function is hidden in rule (1). Consider  $(\neg x)[\neg\lambda y.y/x] \equiv \neg\neg\lambda y.y$ . This is not a  $\lambda^q$ -term by grammar (4.1) but applying the sign concatenation function yields the term  $\lambda y.y$ .

#### 4.3. Reduction

The  $\gamma$ -relation of the  $\lambda^q$ -calculus is of the same form as that of the  $\lambda^p$ -calculus.

$$\gamma^q \triangleq \left\{ \begin{array}{l} \left( [M_i^{i \in 1..m}] [N_j^{j \in 1..n}], [M_i^{i \in 1..m} N_j^{j \in 1..n}] \right) \\ \text{s.t. } M_i, N_j \in \text{LambdaQTerm}, m > 1 \text{ or } n > 1 \end{array} \right\} \quad (4.4)$$

We omit the superscript when it is clear if the terms under consideration are  $\lambda^p$ -terms or  $\lambda^q$ -terms. We still write  $\gamma(M)$  for the  $\gamma$ -normal form of  $M$ .

We extend the  $\beta$ -relation to deal properly with signs.

$$\beta^q \triangleq \left\{ \begin{array}{l} ((S\lambda x.M)N, SM[N/x]) \\ \text{s.t. } S \in \text{Sign}, \text{ and } S\lambda x.M, N \in \text{LambdaQTerm} \end{array} \right\} \quad (4.5)$$

#### 4.4. Evaluation Semantics

We modify the call-by-value evaluation semantics of the  $\lambda^p$ -calculus.

$$\begin{array}{c} \frac{}{v \rightsquigarrow v} \text{(Refl)} \quad (\text{for } v \text{ a value}) \\ \frac{\gamma(M) \rightsquigarrow S\lambda x.P \quad \gamma(N) \rightsquigarrow N' \quad \gamma(SP[N'/x]) \rightsquigarrow v}{MN \rightsquigarrow v} \text{(Eval)} \\ \frac{\gamma(M) \rightsquigarrow v_1 \quad \gamma(N) \rightsquigarrow v_2}{(M, N) \rightsquigarrow (v_1, v_2)} \text{(Coll)} \end{array}$$

#### 4.5. Observation

We define an observation function  $\Xi$  from  $\lambda^q$ -terms to  $\lambda$ -terms as the composition of a function  $\Delta$  from  $\lambda^q$ -terms to  $\lambda^p$ -terms with the observation function  $\Theta$  from  $\lambda^p$ -terms to  $\lambda$ -terms defined in (3.5). Thus,  $\Xi = \Theta \circ \Delta$  where we define  $\Delta$  as follows.

$$\Delta(Sx) = x \quad (4.6)$$

$$\Delta(S\lambda x.M) = \lambda x.\Delta(M) \quad (4.7)$$

$$\Delta(M_1 M_2) = \Delta(M_1) \Delta(M_2) \quad (4.8)$$

$$\Delta([M_i : a_i, b_i, n_i]) = \left[ \Delta \left( M_i^{i \in \{i \mid n_i \neq 0\}} : |n_i| \right) \right] \quad (4.9)$$

Note that unlike the observation function  $\Theta$  of the  $\lambda^p$ -calculus, the observation function  $\Xi$  of the  $\lambda^q$ -calculus is not total. For example,  $\Xi(x, \neg x)$  does not yield a  $\lambda$ -term because  $\Delta(x, \neg x)$  is the empty collection, which is not a  $\lambda^p$ -term.

Although observing a  $\lambda^p$ -term is statistically indistinguishable from observing its  $\gamma$ -normal form, observing a  $\lambda^q$ -term is, in general, statistically distinguishable from observing its  $\gamma$ -normal form.

#### 4.6. Observational Semantics

The observational semantics for the  $\lambda^q$ -calculus is similar to that of the  $\lambda^p$ -calculus (3.9). It is given by a single rule.

$$\frac{M \rightsquigarrow v \quad \Xi(v) = N}{M \multimap N} \text{(ObsQ)} \quad (4.10)$$

#### 4.7. Examples

We provide one example. We show how satisfiability may be solved in the  $\lambda^q$ -calculus. We assume possible solutions are encoded some way in the  $\lambda^q$ -calculus and there is a term  $\text{CHECK}_f$  that checks if the fixed Boolean formula  $f$  is satisfied by a particular truth assignment, given as the argument. The output from this is a collection of  $\underline{\text{T}}$  (truth) and  $\underline{\text{F}}$  (falsity) terms. We now present a term that will effectively remove all of the  $\underline{\text{F}}$  terms. It is an instance of a more general method.

$$\text{REMOVE-F} \equiv \lambda x. \underline{\text{IF}} x x (x, \neg x) \quad (4.11)$$

We give an example evaluation.

$$\begin{aligned}
\text{REMOVE-F } (\underline{F}, \underline{T}, \underline{F}) &\equiv (\lambda x. \underline{\text{IF}} x x (x, \neg x)) (\underline{F}, \underline{T}, \underline{F}) \\
&\rightarrow_{\gamma} \left( \begin{array}{l} (\lambda x. \underline{\text{IF}} x x (x, \neg x)) \underline{F}, \\ (\lambda x. \underline{\text{IF}} x x (x, \neg x)) \underline{T}, \\ (\lambda x. \underline{\text{IF}} x x (x, \neg x)) \underline{F} \end{array} \right) \\
&\rightarrow_{\beta} ((\underline{F}, \neg \underline{F}), \underline{T}, (\underline{F}, \neg \underline{F})) \\
&\equiv (\underline{F}, \neg \underline{F}, \underline{T}, \underline{F}, \neg \underline{F})
\end{aligned} \tag{4.12}$$

Observing the final term will always yield  $\underline{T}$ . Note that the drawback to this method is that if  $f$  is unsatisfiable then the term will be unobservable. Therefore, when we insert a distinguished term into the collection to make it observable, we risk observing that term instead of  $\underline{T}$ . At worst, however, we would have a fifty-fifty chance of error.

Specifically, consider what happens when the argument to  $\text{REMOVE-F}$  is a collection of  $\underline{F}$ 's. Then  $\text{REMOVE-F } \underline{F} = (\underline{F}, \neg \underline{F})$ . We insert  $\underline{I} \equiv \lambda x. x$  which, if we observe, we take to mean that either  $f$  is unsatisfiable or we have bad luck. Thus, we observe the term  $(\underline{I}, \underline{F}, \neg \underline{F})$ . This will always yield  $\underline{I}$ . However, we cannot conclude that  $f$  is unsatisfiable because, in the worst case, the term may have been  $(\underline{I}, \text{REMOVE-F } \underline{T}) = (\underline{I}, \underline{T})$  and we may have observed  $\underline{I}$  even though  $f$  was satisfiable. We may recalculate until we are certain to an arbitrary significance that  $f$  is not satisfiable.

Therefore, applying  $\text{REMOVE-F}$  to the results of  $\text{CHECK}_f$  and then observing the result will yield  $\underline{T}$  only if  $f$  is satisfiable.

## 5. Simulation to quantum computers

We show that the  $\lambda^q$ -calculus can efficiently simulate the *one-dimensional partitioned quantum cellular automata* (1d-PQCA) defined in [11]. By the equivalence of 1d-PQCA and quantum Turing machines (QTM) proved in [11], the  $\lambda^q$ -calculus can efficiently simulate QTM.

To show that 1d-PQCA can be efficiently simulated by the  $\lambda^q$ -calculus, we need to exhibit a  $\lambda^q$ -term  $M$  for a given 1d-PQCA  $A$  such that  $A$  after  $k$  steps is in the same superposition as  $M$  after  $P(k)$  steps, with  $P$  a polynomial.

We assume for now that the 1d-PQCA has transition amplitudes not over the complex numbers, but over the positive and negative rationals. It has been shown [3] that this is equivalent to the general model in QTM.

To express  $A$  in  $M$ , we need to do the following things.

1. Translate states of  $A$  into  $\lambda^q$ -terms that can be compared (e.g. into Church numerals).
2. Translate the acceptance states and the integer denoting the acceptance cell into  $\lambda^q$ -terms.
3. Create a  $\lambda^q$ -term  $\mathbf{P}$  to mimic the operation of the permutation  $\sigma$ .
4. Translate the local transition function into a transition term. For 1d-PQCA this means translating the matrix  $\Lambda$  into a term  $\mathbf{L}$  comparing the initial state with each

of the possible states and returning the appropriate superposition.

5. Determine an injective mapping of configurations of  $A$  and configurations of  $M$ .

Although we will not write down  $M$  in full, we note that within  $M$  are the mechanisms described above that take a single configuration, apply  $\mathbf{P}$ , and return the superposition as described by  $\mathbf{L}$ .

We recall that the contextual closure of the  $\beta^q$ -relation is such that  $M, N \rightarrow_{\beta} M', N'$  where  $M \rightarrow_{\beta} M'$  and  $N \rightarrow_{\beta} N'$ . Thus there is parallel reduction within superpositions. By inspection of the mechanisms above it follows that  $k$  steps of  $A$  is equivalent to a polynomial of  $k$  steps of  $M$ .

Steps 1, 2, and 3 are straightforward. Then for step 5, the  $\lambda^q$ -superposition  $[(M_i : a_i, b_i, n_i)]$  (let  $n = \sum n_i$ ) will be equivalent to the 1d-PQCA-superposition  $\sum \frac{n_i}{n} |c(M_i)\rangle$ , where  $c$  takes  $\lambda^q$ -terms and translates them into 1d-PQCA configurations. Essentially this means stripping off everything other than the data, that is to say, the structure containing the contents. Note that  $c$  is not itself a  $\lambda^q$ -term. It merely performs a fixed syntactic operation, removing extraneous information such as  $\mathbf{P}$  and  $\mathbf{L}$ , and translating the Church numerals that represent states into the 1d-PQCA states. This is injective because the mapping from states of  $A$  into numerals is injective. Thus, step 5 is complete.

Step 4 requires translating the  $\Lambda$  matrix into a matrix of whole numbers, and translating an arbitrary 1d-PQCA superposition into a  $\lambda^q$ -superposition. The latter is done merely by multiplying each of the amplitudes by the product of the denominators of all of the amplitudes, to get integers. We call the product of the denominators here  $d$ . We perform a similar act on the  $\Lambda$  matrix, multiplying each element by the product of all of the denominators of  $\Lambda$ . We call this constant  $b$ . Then we have that  $T = b\Lambda$  is a matrix over integers. This matrix can be considered notation for the  $\lambda^q$ -term that checks if a given state is a particular state and returns the appropriate superposition. For instance, if

$$\Lambda = \begin{pmatrix} \frac{2}{3} & \frac{1}{3} \\ 0 & 1 \end{pmatrix}$$

then

$$T = b\Lambda = 9\Lambda = \begin{pmatrix} 6 & 3 \\ 0 & 9 \end{pmatrix}$$

which we can consider as alternate notation for

$$\begin{aligned}
\mathbf{Q} &\equiv \lambda s. \text{IF } (\text{EQUAL } s1) (1,1,1,1,1,2,2,2) \\
&\quad (\text{IF } (\text{EQUAL } s2) (2,2,2,2,2,2,2,2))
\end{aligned}$$

Then it follows that if  $c$  is a superposition of configuration of  $A$ , applying  $\Lambda$   $k$  times results in the same superposition as applying  $T$   $k$  times to the representation of  $c$  in the  $\lambda^q$ -calculus.

## 6. Conclusion

We have seen two new formalisms. The  $\lambda^p$ -calculus allows expression of randomized algorithms. The  $\lambda^q$ -calculus allows expression of quantumized algorithms. In these calculi,

observation is made explicit, and the notion of superposition common to quantum physics is formalized for algorithms.

This work represents a new direction of research. Just as the  $\lambda$ -calculus found many uses in classical programming languages, the  $\lambda^p$ -calculus and the  $\lambda^q$ -calculus may help discussion of randomized and quantum programming languages.

It should not be difficult to see that the  $\lambda^p$ -calculus can simulate a probabilistic Turing machine and we have shown that the  $\lambda^q$ -calculus can simulate a quantum Turing machine (QTM). However, as we have shown, the  $\lambda^q$ -calculus can efficiently solve NP-complete problems such as satisfiability, while there is widespread belief (e.g. [2]) that QTM cannot efficiently solve satisfiability. Thus, the greater the doubt that QTM cannot solve NP-complete problems, the greater the justification in believing that the  $\lambda^q$ -calculus is strictly stronger than QTM.

It should also follow that a probabilistic Turing machine can (inefficiently) simulate the  $\lambda^p$ -calculus. However, it is not obvious that a quantum Turing machine can simulate the  $\lambda^q$ -calculus. An answer to this question will be interesting. If quantum computers can simulate the  $\lambda^q$ -calculus efficiently, then the  $\lambda^q$ -calculus can be used as a programming language directly. As a byproduct, satisfiability will be efficiently and physically solvable. If quantum computers cannot simulate the  $\lambda^q$ -calculus efficiently, knowing what the barrier is may allow the formulation of another type of computer that can simulate it.

## 7. Acknowledgements

Thanks to Stuart Shieber for helpful comments.

## References

- [1] BARENDREGT, Hendrik Pieter, *The lambda calculus: its syntax and semantics*, North-Holland (1981).
- [2] BENNETT, Charles H., Ethan BERNSTEIN, Gilles BRASSARD, and Umesh VAZIRANI, "Strengths and Weaknesses of Quantum Computing," available online as quant-ph/9701001 at <http://xxx.lanl.gov/abs/quant-ph/9701001>.
- [3] BERNSTEIN, E. and U. VAZIRANI, "Quantum complexity theory," *Proceedings of the 25th Annual ACM Symposium on Theory of Computing* (1993), 11-20.
- [4] CHURCH, Alonzo, "An unsolvable problem of elementary number theory", *American Journal of Mathematics* **58** (1936), 345-363.
- [5] DEUTSCH, David, "Quantum theory, the Church-Turing principle and the universal quantum computer", *Proc. R. Soc. Lond.* **A400** (1985), 97-117.
- [6] DEUTSCH, David, "Quantum computational networks", *Proc. R. Soc. Lond.* **A425** (1989), 73-90.
- [7] HEISENBERG, Werner, *Physics and philosophy*, Harper & Bros. (1958).
- [8] MAYMIN, Philip, "Extending the Lambda Calculus to Express Randomized and Quantumized Algorithms," available online as quant-ph/9612052 at <http://xxx.lanl.gov/abs/quant-ph/9612052>. Many of the proofs omitted from the current paper because of space considerations can be found here.
- [9] MAYMIN, Philip, "The lambda-q calculus can efficiently simulate quantum computers," available online as quant-ph/9702057 at <http://xxx.lanl.gov/abs/quant-ph/9702057>.
- [10] SIMON, Daniel, "On the power of quantum computation", *Proc. 35th Annual Symp. FOCS* (1994).
- [11] WATROUS, John, "On One-Dimensional Quantum Cellular Automata," *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science* (1995), 528-537.